ARPA ORDER NO. 2223

ISI/RR-78-71
August 1978

LEVEL II

Susan L. Gerhart

Program Verification in the 1980s: Problems, Perspectives, and Opportunities

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ISI/RR-78-71
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE *(and Subtitle)*: Program Verification in the 1980's: Problems, Perspectives, and Opportunities
5. TYPE OF REPORT & PERIOD COVERED: Research rept.
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Susan L. Gerhart
8. CONTRACT OR GRANT NUMBER(s): DAHC 15-72-C-0308; ARPA
9. PERFORMING ORGANIZATION NAME AND ADDRESS:
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90291
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
ARPA Order #2223
11. CONTROLLING OFFICE NAME AND ADDRESS:
Defense Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209
12. REPORT DATE: Aug[illegible]978
13. NUMBER OF PAGES: 35
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: ---
15. SECURITY CLASS. *(of this report)*: Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT *(of this Report)*

This document approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

---

18. SUPPLEMENTARY NOTES

Also presented at the conference Oregon Report on Computing: Problems of the 1980s, Portland, Oregon, March 1978.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

program proving, program testing, program verification, programming methodology, software engineering

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Properties of programs can be mathematically proved. This report concerns the use of such mathematical proofs as a means of verifying that programs satisfy their specifications and other expectations of proper behavior. Moreover, the theory by means of which programs are proved can be used in the formal reasoning needed to construct and maintain programs. The primary current needs are: (1) expansion of the theory to encompass more aspects of program correctness, (2) evolution of the theory's mathematical content and form to make it more effective in verifying programs, and (cont.)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

407 952

20.

(3) experimentation with new and current techniques for using the theory in verification and construction, (4) development of human knowledge and skills to fulfill human roles of specifying and guiding program proofs, (5) technological support to take over mechanical parts of the proofs and follow human guidance in elaborating them.

The needed breakthroughs toward the use of program proving as a normal programming activity are: (1) a coherent connection with program testing, (2) evolution of the theory to the point where significant amounts of new program proofs are adapted or reused from previous proofs, (3) development of experimental methodology for effectively evaluating various paradigms and techniques for program proving, (4) greatly increased mechanical theorem proving capacity to reduce the burden on human verifiers, (5) large-scale demonstrations or program proving to evaluate the validity of the activity and to stimulate future research and development.

The ultimate effects of program verification are partly the intangibles of deeper understanding of programs and raising of standards to more closely approximate the theoretical perfectibility of programs. More tangible effects are having formal reasoning methods available throughout program construction (especially applied to software components) and backed up by extensive formal proofs of final products where warranted. Proofs are seen as a necessary complement to the experimental verification provided by testing.

ARPA ORDER NO. 2223

ISI/RR-78-71
August 1978

Susan L. Gerhart

Program Verification in the 1980s: Problems, Perspectives, and Opportunities

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO DAHC15 72 C 0308, ARPA ORDER NO. 2223.

CONTENTS

ACKNOWLEDGMENTS

The ISI Program Verification group has been very helpful in formulating the problems and needed breakthroughs discussed in this paper and in criticizing the final version. However, any distortions or omissions are my responsibility. I would also like to thank Brian Randell for sending me the Turing material.

ABSTRACT

Properties of programs can be mathematically proved. This report concerns the use of such mathematical proofs as a means of verifying that programs satisfy their specifications and other expectations of proper behavior. Moreover, the theory by means of which programs are proved can be used in the formal reasoning needed to construct and maintain programs. The primary current needs are: (1) expansion of the theory to encompass more aspects of program correctness, (2) evolution of the theory's mathematical content and form to make it more effective in verifying programs, (3) experimentation with new and current techniques for using the theory in verification and construction, (4) development of human knowledge and skills to fulfill human roles of specifying and guiding program proofs, (5) technological support to take over mechanical parts of the proofs and follow human guidance in elaborating them.

The needed breakthroughs toward the use of program proving as a normal programming activity are: (1) a coherent connection with program testing, (2) evolution of the theory to the point where significant amounts of new program proofs are adapted or reused from previous proofs, (3) development of experimental methodology for effectively evaluating various paradigms and techniques for program proving, (4) greatly increased mechanical theorem proving capacity to reduce the burden on human verifiers, (5) large-scale demonstrations of program proving to evaluate the validity of the activity and to stimulate future research and development.

The ultimate effects of program verification are partly the intangibles of deeper understanding of programs and raising of standards to more closely approximate the theoretical perfectibility of programs. More tangible effects are having formal reasoning methods available throughout program construction (especially applied to software components) and backed up by extensive formal proofs of final products where warranted. Proofs are seen as a necessary complement to the experimental verification provided by testing.

"It is of course important that some efforts be made to verify the correctness of assertions that are made about a routine. There are essentially two types of method available, the theoretical and the experimental. In the extreme form of the theoretical method a watertight mathematical proof is provided for the assertion. In the extreme form of the experimental method the routine is tried out on the machine with a variety of initial conditions and is pronounced fit if the assertions hold in each case. Both methods have their weaknesses."
--Alan Turing, circa 1950, *Programmers' Handbook for the Manchester Computer*

1. INTRODUCTION

Before looking into the future of program verification it is worth reconsidering Turing's advice of nearly three decades ago, especially regarding the weaknesses of the two extreme positions. The theoretical approach must deal with the fact that very few mathematicians ever carry out a proof down to the last detail of axioms and rules of inference because the process is simply too exhausting for both writer and reader and is still prone to error. (Unlike Turing, we can envision the possibility of mechanizing much of the detailed proof effort, although this is one of the hardest problems of all.) The experimental approach must address the question, "By what sound argument can you claim that the program will satisfy the assertions when executed on data which was not part of the experiment, even if the program executed perfectly for all data within the experiment?" The only possible answer is another mathematical argument. (Surprisingly, there has been little real progress in either discovering or disproving the existence of such arguments.)

Given two flawed extreme approaches, what does one do? In the *Programmers' Handbook for the Manchester Computer*, Turing recommends the ever-popular, but hard, "desk checking"--hand checking conscientiously while summarizing and organizing via "check sheets." He also argues for having another programmer complete a reduced version of the check sheets and for deliberately forgetting the purpose and method of a routine to avoid missing program errors due to preconceived ideas. He counsels against making alterations in the middle of the routine without verifying that the earlier parts are unaffected, and recommends explicitly checking by program that input assumptions are satisfied. Turing claims that most errors will be found by such thorough checking, but also cites an example where the probability of selecting the right case to reveal a particular error was 2**-10. It is also recommended that the state of the machine be described by mathematical expressions, in order to convey the "theory of a routine." His

overall approach mediates the two extremes: experimental in the use of carefully selected actual data and theoretical in the use of human reasoning to meticulously check for errors and to make assertions which display the underlying theory and provide the guide for checking the program.

This quotation and summary are useful for starting to think about the future of program verification for several reasons:

1. They show more common sense motivation, technique, and caution than most modern textbooks or programmer handbooks, which view verification as either testing or proving, or which don't discuss it at all.

2. As in Turing's time, common sense tells us that taking any extreme position is fraught with potential disaster. The most rational course is moderate, combining the strengths and avoiding the weaknesses of the two methods. To dramatize the point, consider your feelings upon stepping onto an aircraft piloted by a completely computerized aircraft control system. How would you feel if told the software had never been tested but had been thoroughly proved? Or that it had been tested according to the latest standards--say at least by executing every statement and a wide variety of conjectured conditions--but never exposed to a mathematical argument concerning untested conditions? Given the few current demonstrations of practical proving, we would probably feel safer about the latter. But if we investigate far enough to discover the inadequacy of the current theory behind testing and if we remember the surprises that continually follow upon the release of "thoroughly tested" software, we should also have some misgivings about testing alone. Most of us would rationally prefer that correctness had been strongly argued for all data and had been fully demonstrated for considerable data. We might also demand that the entire computer system--hardware, software, and human operators--be justified in terms of probabilistic reliability arguments.

3. But for the sake of scientific study, we must choose one of the extreme points and investigate it thoroughly in its separate context: develop its theoretical basis; explore a variety of paradigms and techniques for performing its associated activities; subject these paradigms to experimental investigation to evaluate their feasibility, practicality, and applicability to various types of problems; identify and acquire the skills and tools for performing the experiments and eventually pursuing the activity in practice; and identify, understand, and accept its overall strengths and weaknesses.

The present discussion is meant to be followed in the light of common sense: *neither* proving nor testing warrants full confidence as *the only* method of program verification. Ultimately, the best course may be (1) some combination of mathematical arguments with testing, (2) application of one or the other methods

in recognizably acceptable situations, or (3) perhaps the simultaneous performance of the two activities independently but in moderation. Before any of these three combinations can be properly investigated, there must be a greater understanding of program testing than currently exists; ideally we would like a real theory. This further justifies the study of program proving as not merely our only current way to reason formally about programs but also as a stimulus to a theoretical basis for testing.

To summarize, in our context program verification (hereafter abbreviated as PV) means mathematical proof of the *consistency* of the program with assertions about it, the external assertions usually being called "specifications." The definition emphasizes consistency, recognizing that specifications must separately be related to the full demands of the user. However, we will follow current terminology in loosely referring to consistency as "correctness" and the associated activity as "program proving." The current view of the goals of PV is to attain a high degree of confidence that the program satisfies its specifications relative to given semantics of and assumptions about the software, hardware, and user environment and to the absence of errors in the verification process. Of course, this is no *certainty* at all, the environment being almost impossible to describe completely and the process being error-prone. The confidence comes from having a theoretically sound and systematic method for arguing consistency, tools and skills for carrying out the process, dedication of sufficient resources for its completion, and various checks for errors during the process. Having all these stimulates the improvement of the environment in both fact and description so as to reinforce the confidence gained by formal reasoning.

PV has an important payoff besides verification. It takes a view of programs as mathematical objects and requires a theory about them. As we shall see, this theory suggests that programs can be classified and then studied by class, for example by task or technique; shows that there may be certain structures and relations between these structures that generate a large range of programs independent of their assigned task; rejects programs which are unduly disorganized or which fail to display their underlying organization; and begins to explain the complexity and difficulty of programming and to suggest ways of surmounting these. In other words, PV demands a thorough and basic understanding of programs, including language and design principles, that may ultimately strongly influence their construction. Many researchers now view this theory as the central reason for studying PV, a view which will be emphasized in this report.

2. CURRENT PROBLEMS, OBSTACLES, AND STEPS TO OVERCOME THEM

We will explore two approaches to understanding the current state of PV: (1) an enumeration classified as theory, technique, people, and technology and (2) a historical and psychological analysis which reveals sources of confusion, tension, and conflict which block a clear view of what is happening and what should happen. Other surveys of program verification are [London77a,77c] and [Luckham77].

The mathematical theory of programs is multilevel. A first level relates statement and declaration language constructs to predicates. All current formalisms look something like that popularized in [Dijkstra76] as the "predicate transformer" wlp(S,Q), the "weakest liberal precondition for a state to satisfy in order that the predicate Q be satisfied for the state resulting from executing the statement S." This gives P⊃wlp(G,Q) as the correctness theorem for a program G with specifications P for input and Q for output. A second level deals with the expressions of the language, requiring some axiomatization of the data types of the language. A third level is the assertion language, which is usually a superset of the expression language. Cutting across all three levels is some language of logic, including means for expressing quantification, equality, implication, etc. For example, denoting the predicate transformers for the conditional and assignment statements by

$$\text{wlp(if B then S fi, Q)}=(B \supset \text{wlp(S,Q)}) \wedge (\sim B \supset Q)$$
$$\text{wlp(V:=E, Q)}=Q[\text{E substituted for free occurrences of V}]$$

and letting t be a binary tree and S a stack,

$$\text{wlp(if Right(t)}\neq\text{nil then S:=PushStack(S,Right(t)) fi,}$$
$$\text{NoNilNodesOnStack(S))}=$$
$$=(\text{Right(t)}\neq\text{nil} \supset \text{NoNilNodesOnStack(PushStack(S,Right(t))))}$$
$$\wedge(\text{Right(t)}=\text{nil} \supset \text{NoNilNodesOnStack(S))}$$

where NoNilNodesOnStack(S) is some predicate defined for stacks.

These are the types of theorems, sometimes called *verification conditions* or *verification lemmas*, that are usually subjected to detailed proofs in current techniques.

Another level of theory deals with the way data types and statement level theories should be organized to facilitate efficient theorem proving, which we will discuss further under technology. Of course, any individual program proof also requires the theory associated with its problem domain.

There are many technical issues dealing with the theory. The kinds of correctness which can be addressed range from "partial correctness," which ignores termination questions, through various degrees of termination, e.g., "cleanness" in not aborting on inexecutable operations and/or "nonlooping," to inclusion of performance constraints. Current theories lean toward partial correctness with the question of nonlooping handled by bounded counters or well-ordered functions. Cleanness is addressed by treating the conditions for proper execution of operations like the input specifications of functions. But exceptional conditions, including errors, in user input, other software components, and even hardware are aspects of correctness that have yet to be adequately addressed.

Theoretical Problem 1: Theory must address requirements far wider than "partial correctness," including recognition and handling of abnormal conditions.

This is not simply a verification problem, since languages and methodology have only recently considered it seriously. Correctness theory may be especially useful in guaranteeing robustness and handling exceptions, since it already formally describes how properties are affected as they flow through the program, including properties relating to exceptions. Steps in this direction are [Levin77] on exception handling, [Pratt77] on separating various nondeterministic termination issues, [Lampson77] with "legality assertions" for proper execution of expressions, and [Suzuki77] on checking array bounds.

Any good theory must have the qualities of "soundness," i.e., that all theorems are valid in the desired sense, and "completeness," i.e., that all true and interesting statements can be proved. For a correctness theory, the question is whether the theory fully and accurately captures the semantics of the associated language. In this respect great progress has been made in developing a general theory of semantics as the foundation for program proving [deBakker77] and in considerable informal experimentation with proof rules for various languages and individual language constructs [Hoare73, London77b]. However, current languages have complicated proof rules which are hard to justify semantically.

Theoretical Problem 2: Current theory must be stretched to include more features of real languages and, concurrently, languages must be designed which admit reasonably simple and complete semantic descriptions.

Jovial is addressed successfully in [Elspas77], concurrent programming constructs are axiomatized in [Owicki76], pointers and records are handled in [Luckham76], and microprograms of significant complexity are handled in [Carter77]. New languages with a full complement of necessary features are being developed in conjunction with some degree of formalized semantics and goals of verifiability [Ambler77, Lampson77, Liskov77, Wulf76], but there will be a gap for several

years between theory and current languages, e.g., PL/1, BLISS, etc. It is unlikely that this "clean-up" operation on languages would have occurred as urgently without the motivation and formalism of PV. However, it may be that proving the language-independent aspects of algorithms and data structures will achieve most of the benefits of verification and that language-dependent questions may be resolved by proving equivalence of programs or by relying on compiler-like checks.

One of the deep and most difficult parts of mathematics is mathematical induction, which allows reasoning about the structure and properties of potentially infinite objects. Correctness theory started off with the notion of "invariants," assertions that hold at every iteration of a loop. Another early inductive form of argument was based on the "structure" of objects [Burstall69]: assuming a property for its components, the property was proved for an arbitrary instance of a well-defined object. More recent variations are subgoal induction [Morris77], which reasons with invariants both forward and backward, and intermittent assertions [Manna76], which reason from one iteration of a loop to some future, but not necessarily the next, iteration. The question is whether it makes any difference which inductive method is chosen and, if so, when. As in traditional mathematics, it is critically important to get the statement of the theorem exactly right for an induction argument, after which the argument usually goes through smoothly.

> *Theoretical Problem 3: More insight is needed into the basic nature of inductive arguments in program proving, when to use which type of argument and how to formulate the inductive assertions.*

Some work has been done on mechanical generation of inductive assertions from programs and from specifications [Wegbreit74], but the problem is so hard that these approaches may work only in very simple instances. Another approach is to accumulate inductive assertions during program derivations as they are made in a specific context for a specific purpose at a useful level of abstraction, rather than all at the end.

The predicate calculus is not the most understandable language, either to casual readers or fluent mathematicians.

> *Theoretical Problem 4: Ways must be found to improve the expressibility and readability of formalisms, both at the semantic and at the assertion level.*

The algebraic approach [Guttag76, Burstall77b] promises some alleviation by orienting reasoning toward equalities. The key notion may be simply the right notation for the concepts of interest.

Mathematicians never use a theory at its axiomatic level for very long. Instead, they develop general and leading theorems which express the important properties of the concepts of the theory and which suggest new variations and derivatives of these. Program proving is currently like reading the first chapter of a mathematics text to find the axioms and basic definitions and then trying to do the hard exercises at the end of the book. Because the intermediate theory is missing, each exercise requires building up intermediate concepts and techniques or brute force applications of axioms.

Theoretical Problem 5: The theory of programs based on correctness properties must evolve to higher levels of generality in both theorems and techniques.

An active area of theory building, at least at the axiomatic level for data structures, is [Guttag76], leading possibly to an overall theory of the structure of such axiomatic theories [Burstall77b]. Further examples of this higher level theory will be given shortly. While not yet highly visible, this type of theory is the key to making PV follow the successful route of mathematics, namely pursuit of stimulating and general statements about the objects (here programs) of interest.

Turning from theory to those techniques that use the theory for proving programs, the basic question is what theorems to prove. The current paradigm has two disjoint phases: (1) to completely transform assertions into predicates containing only expressions, not statements, from the program and then (2) to prove these predicates. This paradigm, though conceptually simple, has numerous practical difficulties: the generated predicates may lose all traces of structure from the program, thereby foiling attempts to find a proof of them based on reasoning about the program; recovery from an error, no matter how minor, in either program or assertions is almost impossible without a complete restart; and the "calculus of programs" cannot be brought into play. This calculus consists of derived properties of predicate transformers, e.g.,

$$wlp(S,A\wedge B)=wlp(S,A)\wedge wlp(S,B),$$
$$(P\supset wlp(S,Q))=(sp(S,P)\supset Q),$$

where sp is a "strongest postcondition" transformer.
These properties allow considerably more flexibility in organizing proofs and are perfectly legal ways of proving programs, although excluded by the two-stage paradigm.

It is also possible to prove concrete programs correct by virtue of their being instances of more abstract programs. The appendix contains an iterative version of a program schema that is proved to compute a recursive function, stated in terms of abstract operations F, h0, h1, h2, h3, G, where only the definition of F and the associativity of G are known. This schema can be instantiated to variations of tree

traversal and perhaps to other functions. One proof at the schema level is used for two proofs at the concrete level with the further gain that schema level proofs, freed from the distraction of concrete operations, are easier to find and understand.

At another level are theorems which allow correctness to be transferred from one program to another without complete reproof of the second program. A Context Theorem informally says that the correctness of an entire program is equivalent to proving the correctness of every statement of the program in a "context" which describes its "strongest possible precondition" and "weakest necessary postcondition" relative to the specifications and assertions in the rest of the program. This supports a Replacement Theorem which informally says that one statement S can be replaced by another S' in a program G to give a program G' that will be correct if G was correct and if S' is correct in the context of S within G. The replacements can often be generalized and partially proved, leaving just a residue of proof related to the problem domain. These theorems lead to a paradigm for proving programs correct by successive transferral along a chain of replacements; it will be illustrated in the appendix. The "correctness-preserving transformation" paradigm is being actively pursued in, among others, [Balzer76 Burstall77a, Gerhart75].

Yet another theorem supports the use of Ghost (or Auxiliary) Variables [Gerhart78], which have no influence on the result of the program, but which are useful to express histories, used-up values, or missing abstractions. The theorem says that a program proved correct using these as regular program variables is still correct when the variables are deleted from the program and appear existentially quantified in the assertions on the proved program. Schemas often use an abstract data object from which some other value or data object is computed in an instantiation of the schema, after which the abstraction is removed. In the appendix example, the tree traversal computes a list of nodes of the tree upon which a count is made. Only the count is of interest, so the nodelist is deleted.

> *Technique Problem 1: The two-stage generate-and-prove verification lemmas approach must be relaxed to admit natural higher level proof techniques and preproved programs. The variant paradigms must be clearly formulated, formally justified, and subjected to experimentation.*
>
> *Technique Problem 2: Proving more general theorems reduces overall proof effort, but what are the useful theorems and what are the right levels for their statements and proofs?*

Our example illustrates these techniques and their problems. The data structure abstraction methodology addresses these problems, but the full range of paradigms and theorem power has not yet been reached.

There are some programming languages which emphasize functions, e.g., LISP, but most languages favor iteration as the main form of looping and sequencing of statements as the main form of composition. Some experience shows that it is much easier to prove properties about functions because they generate modularity, discourage unnecessary sequencing by parallel evaluation of arguments to function calls, and direct attention toward the objects being computed. There is no loss in practice if the proved functions can be translated to iteration and the proofs can be transferred.

> *Technique Problem 3: Which is better, function or iteration representation of programs? How can they be used interchangeably?*

The argument for functions is given by [Manna77, Boyer75], while [Dijkstra76] argues for iteration.

Given the fallibility of programmers and provers, an interesting problem is

> *Technique Problem 4: How much of a proof is still valid after certain types of modifications to the program or to assertions?*

The only work in this area so far is [Moriconi77].

Constantly plaguing those who publish in PV and try to write proofs is

> *Technique Problem 5: How does one present a proof with sufficient structure that remaining details can be filled in, but without so much detail that readers are overwhelmed?*

[Wegbreit77] suggests various means of "justifying" proofs as normal program additions and annotations.

Programs are created by people who should know enough about them to create proofs. But this is not so when rigorous mathematical standards are applied, or even when loose informal arguments are acceptable. Creating a proof requires the ability to design notation that captures the basic concepts, knowledge of rules of logic for sequencing steps of a proof, and an understanding of the axioms that describe the objects of interest. A basic grounding in mathematical knowledge is inescapable, but also necessary is the creative component which is not taught in elementary math courses.

> *People Problem 1: Potential program provers must not only be taught mathematical facts and techniques, but must also be led to develop their manipulative skills and creative powers.*

[Dijkstra74] presents an enlightening discussion of this problem.

Any new formalism or formal discipline is criticized as being an unnecessary academicians' toy for which no practical use can be foreseen and which is just too complicated for ordinary people to understand. BNF was probably seen in this light, especially when followed by a flurry of papers about syntax and parsing. Yet it is now widely accepted and taught without mystery, its use having been found and separated from the formalism which refined it.

> *People Problem 2: The rebellion against formalism and the excess of formalism in early stages must be accepted as normal.*

It may take a while for people to accept the fact that "programming *is* a discipline of a mathematical nature" [Dijkstra74].

Programs can be proved without any type of mechanical support, just as mathematicians have proved theorems for hundreds of years. But program proving differs in that it requires many small and not always interesting theorems. As argued and illustrated above, there are metatheorems which support interesting proof techniques and general theorems which implicitly define interesting classes of programs, but at the concrete level there are always many little problem domain facts and tricky chains of reasoning needed to glue together a proof. Some computer assistance is needed both in handling domains where people don't think well (e.g., integer arithmetic and chains of inequalities) and in making sure every step of a proof is legitimate. Some initial capability in this area exists in present PV systems, but there is no overall theory of how to handle either the large number of disparate domains (basically one per program data type) or the extremely large search space that can be generated in finding proof steps.

> *Technological Problem 1: It appears necessary to find specific domains that are highly useful in PV and concentrate on increasing potency within them. But an overall theory of handling these domains is still needed.*

Steps in these directions are [Nelson77] with coordinated fast simplifiers, [Lankford77] on the properties and use of rewrite rule systems, and [Boyer77b] on the use of lemmas and induction generalization techniques.

As mentioned earlier, the generate-and-prove paradigm is conceptually clean and the generate phase is easily implemented, provided the language semantics are given and clean. But the paradigm fails in practice, and though newer ones are known, they are not yet implemented.

Technological Problem 2: PV systems must be extended to permit use of the "calculus of programs," transferral of correctness between programs, and hierarchical development of programs.

Theory and implementations of some of these ideas appear in [Good77, Elspas77, Musser77, Gerhart76].

PV systems are systems in the true sense, containing the semantic analogs of compilers in predicate transformers, multiple theorem provers working in cooperation and possibly in parallel, input and output routines to manage data in user terms, data bases of previously proved formulae, bookkeeping for the status of proofs, etc. As such they suffer the usual trauma of complexity and bear the additional burden that proofs must proceed interactively without confusing or boring the user. Neither humans nor systems can manage the task of PV alone; the effort must be cooperative and synergistic.

Technological Problem 3: PV systems must be made habitable before they can be experimented with seriously, let alone be put into production use.

Of course, there is the fundamental dilemma: How do you know the verifier is correct?

Technological Problem 4: PV systems must be constructed so clearly and so well that their correctness can be accepted after sufficient periods of reliable usage.

Having looked at various problems and steps regarding the theory, techniques, people and technology of program proving, we will now explore our second approach. While not the type of pure scientific analysis we might like to see in a paper on the future of a scientific activity, the following historical and psychological analysis is still important in determining the course of the field. Researchers are seldom as objective as they might like, but--as in everyday life--find it hard to identify all the determinants of their actions. Understanding the trends in human terms may make it possible to break out of ruts and make better intermediate decisions about what problems to tackle next.

In the development of a theory , there is first a pre-theory stage where some activity goes on guided only by intuition and common sense until someone formulates axioms and rules of inference which provide the language and reasoning mechanisms for discussing the activity and which generate the true statements of interest. Next some useful and general theorems are proved from which more specific, interesting theorems can be proved without recourse to the axioms. Concurrently comes the development of proof techniques that telescope proof steps or systematize reasoning. Some time later key theorems are recognized,

and proofs begin to be organized around certain similar themes that eventually become standardized throughout education and research. Finally, some new problems or insights generate doubts and major revisions.

Related to this progression is the way we view objects and the styles of manipulation performed upon them. First there is the concrete view and "blind," ad hoc manipulation. Then certain patterns evolve which cover most situations and lead to disciplines, still at the concrete level. With sufficient understanding of patterns of manipulation and of similarity of structures comes the ability to generalize from details and thus reach more abstract levels. As abstraction becomes more widely appreciated, it comes to be used before details are considered, though indecision, lack of clarity, and old habits of thought make its use inconsistent. Eventually the emergence of patterns of abstraction leads to disciplines and ultimately to some standardization.

From yet another viewpoint, consider the way problems emerge and are dealt with. Problems may be recognized informally for a long time before the right concepts and terms are found to describe them. Then comes the formalization of former informal techniques as well as new techniques which follow logically from the newly recognized concepts. These are explored haphazardly until some set of critical questions can be posed and some experimental methodology developed. With experimentation, strengths and weaknesses and areas of applications are clarified. Somewhere along the line, techniques become useful enough that dissemination occurs throughout and even outside of the research community. Eventually, a few standard approaches emerge, but by then new problems are being recognized relative to old problems, the new solutions, or externally generated new problems.

The overall effect of these progressions is considerable confusion, tension, frustration, and conflict. In the computer field, these are compounded by the rapidity of developments and by economic and social pressures to provide nearly instantaneous solutions to barely understood and grossly underestimated problems. Consider some observations about the current problems of PV in the light of these progressions:

1. In programming methodology, the move toward abstraction occurred at almost the same time as the move toward concrete discipline. On the one hand, people believed that sticking with three fundamental control structures would solve many problems, while they were being told that abstraction (whatever that was) was the way to manage complexity. There was the conflict between discipline at one level and ad hocness at another. PV is at a similar transition stage: some gain has been made at the concrete level, while abstraction promises much more, at the temporary sacrifice of familiarity and some acquired discipline.

2. When trying to apply a new theory which exists only at the axiom level, the practicing prover soon notices that many mathematical properties and techniques get rediscovered or redone and that starting from scratch on every problem is excruciatingly slow and painful, as well as stupid. But, if provers start consciously developing the theory in a systematic fashion, the amount of notation, theorems, and techniques soon becomes disorganized and unmanageable (witness the size and complexity of mathematics books). Furthermore, there are still so many programs to be proved that each new one can find very little well organized old material to lean on. Our earlier analogy with proving the hard exercises at the end of the book given only the first chapter encapsulates the dilemma of PV, which only time and concentrated effort can cure. Notice that the above-mentioned theorems are really formalizations of common informal reasoning.

3. Related to the previous observation is doubt about the ability to scale up from the small and simple problems used to develop theory and techniques to the messy real problems. At some point, a great deal of effort must be expended in just this scaling up, forcing theory development and introducing new problems of size and interaction complexity.

4. Continually disturbing is the "oversell/overbuy" phenomenon. The new solutions look more promising than the old ones before there is adequate recognition of the new problems that will accompany the new solutions. PV has been accepted by large portions of the research world as a major theme for viewing and attacking all kinds of problems. At least some significant portion of the practicing programming world recognizes its existence and uses it in some diluted form or is affected by such byproducts as newer and cleaner languages. But it is difficult to measure the value of these byproducts and attitude changes relative to the ultimate (but as yet unachieved) goal.

5. It takes a long time to develop adequate experimental methods to evaluate various proposed solutions or paradigms, not to mention the time it takes to perform the experiments. Consequently, the evaluations are hasty, unquantified, and subjective. Therefore, there is always further doubt about validity and usefulness which never has time to be either objectively dispelled or supported. PV, as well as many other parts of computer science, lacks the experimental methodology for conscientiously evaluating its competing methods and validating its claims.

6. Technology for assisting manual techniques always lags far behind because methods must be shown to have some theoretical validity and be somewhat formalized before they can be incorporated into languages which can then be meshed with previous technology. Furthermore, it simply takes a long time to develop comfortable and reliable technological support. This also goes along with

the bottleneck of experimentation: in order to do any large scale experimentation, technology is necessary, but it is difficult to know how to direct the technology before usefulness has been demonstrated. In PV, the gap between methodology and technology must be filled by languages, although some new paradigms may be explored at the concrete level and with conventions for current languages.

7. One further complication is that as soon as some solution looks promising, its earliest version is disseminated. Of course, this early version has difficulties which cause frustrations and doubt; even if it gets mastered, ingrained habits and inertia may cause improved methods to be rejected as they come along. The invariant assertion method associated with full verification condition generation on concrete programs has now been widely written up and is frequently taught. While this conveys some of the fundamental ideas of PV, the inflexibility and distasteful separation from program construction make it unduly difficult in practice and misrepresent the difficulties of PV.

8. Since everyone learned programming early in her (or his) computer science career and usually got along well enough fc several years with that amount of knowledge, there is reluctance to accept the unpleasant fact that there is always so much more to learn, especially when it means breaking years of habit. PV confronts programmers with their lack of understanding of programs and their inability to express what they do understand. Sometimes the challenge is accepted and proving is mastered, but sometimes the frustration at not understanding gets displaced to proving rather than programming.

These eight points all relate to the compression of many stages of research and development into a very short period of time and to our basic human tendencies to expect too much too soon for too little effort. While not "solvable" in any sense, they can explain problems which must be endured without distraction from the ultimate goals.

3. NEEDED ADVANCES AND BREAKTHROUGHS

Breakthrough 1: A theory which unifies testing and proving or selects between them.

Since ultimately verification will consist of some combination of testing and proving, it would greatly help both camps to have better perspective. The breakthrough will probably have to come from the testing end of verification, because program proving researchers see their approach, which is based on a rapidly developing mathematical theory, as more promising. The considerable amount of research on sampling and probabilistic testing has neither convinced nor interested the program proving community. The cost of efforts in this direction

could be small, since the breakthrough would most likely come from some ingenious twist on current research to yield the critical insight. Some unifying steps are [Goodenough77, Howden77], which emphasize errors and their relation to both testing and proving. The important factor is that researchers be receptive to, rather than at war with (or, even worse, ignoring) the other camp.

Breakthrough 2: A significant increase in the power of mechanical theorem provers.

Technological improvements in speed and capacity of computers will help, but the real need is for a theory that unifies various strategies in such a way that small but important domains can be well handled. This is one area where optimistic projections for full mastery of the type of theorem proving we want today, namely interactive guidance by user-supplied strategies through fully mechanized subproofs, are *ten* years, with full capabilities for finding proofs, although not necessarily finding interesting theorems, in *thirty* years. There are really only two schools of present theorem proving: the resolutionists, who haven't yet seriously considered PV applications, and the nonresolutionists [Bledsoe74], who have made a major, but ad hoc, attack on PV-related problems. Considerably more research funding could go to this area, but there are not yet many researchers capable of, or interested in, attacking such a hard, long-term problem. It requires a unique combination of mathematical and programming depth of knowledge and experience. A clean theory of mechanical theorem proving will require good implementation to be effective, while good programming skills must be backed up by sound theory and deep insight. However, mechanical theorem proving is not necessary to achieve many of the benefits of program proving; it is necessary only for the highest attainable degree of certainty.

Breakthrough 3: More than one large-scale demonstration of PV.

Even as a combination of manual and mechanical proofs, such demonstrations would command attention and give momentum to PV. Given the current state of technology, skills, and basic knowledge, this seems possible with sufficient dedication of resources. The number of snags will probably be large, but these will suggest many new and interesting research and development tasks. A failure can be attributed either to basic flaws in the approach or to underestimation and undercommitment of resources, either of which would still provide important impetus in the direction of verification efforts. The costs are the dedication of sufficient human and computer time and energy, perhaps at the expense of other theoretical and practical developments.

Breakthrough 4: Development of sound experimental methodology for evaluating various PV paradigms.

Such methodology could clarify and speed up the evaluation of vague paradigms This might be nothing more than the amount of real and computer time required to push through a variety of examples on an existing system, together with the quantity of transfused knowledge and strategic direction provided by users. Unfortunately, current systems are not powerful enough to handle multiple paradigms, and cross-comparisons between systems would be difficult. Nevertheless, even if the experiments cannot be performed for lack of apparatus, the formulation of the experiments and their limited manual application could be valuable. The cost of such an effort would primarily be associated with some coordinating body which selected paradigms, designed experiments, and evaluated results from several projects.

> *Breakthrough 5: The accumulation of sufficient theoretical results to reach a critical mass where new program proofs can reuse significant portions of previous proofs.*

Program proofs which start from scratch, as currently, will be prohibitively expensive, if not completely unmanageable. The main problem is translating current informal knowledge about programs into theoretical terms and organizing this mass of knowledge so that it may be studied and mechanically accessed. This should accompany the normal growth of PV, but it may be possible sometime in the next decade to seriously concentrate on this problem. Our confidence in PV arises from this combination of widely accepted and used higher level theorems with mechanized lower level proof checking, mediated by human creativity in organizing proofs.

> *Breakthrough 6: The management of complexity of PV systems and the design for synergistic human-machine interaction.*

The complexity problem exists for current systems which are nowhere near their end goals. One system which works extremely well for users other than its designers will show the direction for other PV systems. The sustained support of present PV projects, which are well aware of this problem and headed in this direction, should be sufficient, especially when more users gain access to the systems.

4. EFFECTS OF PV SOLUTION/BREAKTHROUGHS ON COMPUTING IN THE 1980s

The effects of PV must be separated into the tangibles and intangibles.

> *(Intangible) Effect 1: The education of programmers can be vastly improved and accelerated.*

Those of us associated with education constantly see the improving quantity and quality of material taught in courses. For example, basic data structure material that was unknown, in a systematic way, by Ph.D. graduates of the early 1970's is now taught routinely to college freshmen. The impact of PV may be to sort out this basic programming material, organize it more systematically, and present it more coherently. Programs with invariant assertions are no more, and probably less, mysterious than unasserted, and often unspecified, programs. It is likely that the textbooks of the 1980s will routinely use program proving ideas without any special fuss about verification. Then more people will be able to read and perform proofs, thereby increasing the validity of PV. In light of the productivity variations between programmers, simply teaching more programmers more material should improve the overall quality of programming.

(Intangible) Effect 2: The standards of quality in programs and programmers should improve.

There can be little doubt that software quality has been low both because little more was expected and because low quality was acceptable. PV makes very high demands on the quality of programs and reveals deficiencies. If at least some "perfect" programs, in every sense of their quality, can be produced and widely disseminated, then perhaps the quest for perfection will be more broadly sought, especially if perfection turns out to pay off. An example in this direction is the Unix operating system, which has gained widespread use simply because it is clean, comfortable, and reliable, if not all-purpose and fancy.

(Tangible) Effect 3: The construction and maintenance of programs will rely greatly on formal reasoning, although not always formal proofs.

Program proving as pure verification separated from construction will probably disappear. This may leave testing as the primary mode of verification, both as confirmation and exhibition of program quality. If maintenance--that is, the fixing of deficiencies as they are recognized plus the adjustment of function to meet new demands--is as expensive as figures seem to indicate, then PV may pay off most here. Even if proofs are not performed or are not reliable in the sense of verification, they demand that programs be fully specified and fully documented in the form of assertions. Such assertions may guide a new form of maintenance, i.e., systematic modification preserving correctness as stated in assertions. As pointed out in [Balzer76], it may be feasible to shift maintenance from the level of concrete code up to appropriate abstract levels, reimplementing when modification is necessary.

(Tangible) Effect 4: Significant sized programs will be proved, albeit at considerable expense.

We will probably see some programs of hundreds of lines of code selected from real applications being proved before 1980. For programs intended for wide distribution and critical applications, the expense will be fully justifiable and will have to be borne. The challenge will be to reduce the expense for full-scale proofs and to appropriately compromise expense with uncertainty for scaled-down proofs.

(Tangible) Effect 5: Standardized components will be built and verified and used.

The long-term goal of software engineering includes the development of off-the-shelf components for many common tasks. This has required specification techniques so that components can be selected and composed, verification techniques so that the components can be trusted, theory and experience to show what should be standardized, and adaptation techniques so that components can be used in many ways in many environments. Theory and practice seem to be reaching the point where this goal can be a reality.

Of course, the ultimate effect may be chaos or destruction. That we can verify software to a high degree of certainty does not mean it will fit well into human social and economic systems. One cannot help but feel the accelerating use of computing systems: electronic fund transfer, maintenance of power facilities, monitoring of real time systems, electronic mail, speech communication, home computers for everyone's daily activities, hospital patient monitoring systems, data bases, etc. While computer scientists cannot solve the issues involved with the disparate uses of computers, perhaps we should try to integrate our painful experience with fallibility with the social and economic systems that will use our results.

Appendix

Example: Variations of Binary Tree Traversal

Purpose: This example is intended to show an overall generality and variety of techniques greater than commonly seen in the literature on program verification. It emphasizes aspects of the correctness-based theory of programs described in the paper, specifically several types of theorems:

(1) Schemas (partially interpreted programs) for an iterative version of a special recursive function and the interpretation of this schema to the more specific, but still generally described, task of preorder tree traversal. For the purpose of comparison with previous publications [London77a, Burstall74], the trees will be traversed to count the "tips" and the "leaves."

(2) Several transformations which allow alteration of conditional statement and loop structure.

(3) A "ghost variable" theorem which allows deletion of a variable used in the schema once it has been related to specific reasons for performing the traversal.

Associated with these theorems are natural methods for proving programs:

(1) By instantiation of proved schemas to concrete programs

(2) By transformation to transfer correctness, with little additional proof effort, from one form to another one more desirable for non-correctness reasons, say optimization or implementation within a restricted set of language constructs.

These higher level methods all rely on the old, familiar invariant assertion method, but partially shift its use to the schema and transformation level. In addition, by following this paradigm of instantiational and transformational proof, we are able to conjecture some possible "laws" which govern the forms of assertions.

Version 1: Schema for iterative version of a recursive function

Let

F(x)= <u>if</u> p(x) <u>then</u> h0(x) <u>else</u> G(G(h1(x),F(h2(x))), F(h3(x)))

where G is an associative operation with identity IG.
For ease of reading, we will write G as an infix operator, symbolized by ⊗

F(x)= if p(x) then h0(x) else h1(x) ⊗ F(h2(x)) ⊗ F(h3(x))

An iterative program for this function is

```
declare S:Stack (with the usual operations
                 PushStack,PopStack,TopStack,CreateStack);
declare x: type D1 with operations h2,h3 producing results of type D1
                 and operations h1,F producing results of type D2
                 and ⊗ an operation on type D2 producing a D2
Acc:=IG; x:=x'; S:=CreateStack;
loop as  F(x')= Acc ⊗ F(x) ⊗ UnravelStack(S)
while ~p(x) or S≠CreateStack do
 if ~p(x) then
   Acc:= Acc ⊗ h1(x);
   S:=PushStack(S,h3(x));  x:=h2(x)
 else
   Acc:= Acc ⊗ h0(x);
   x:=TopStack(S);  S:=PopStack(S)
repeat;
Acc:= Acc ⊗ h0(x);
assert F(x')=Acc
where UnravelStack(S)=
if S=CreateStack then IG  else F(TopStack(S)) ⊗ UnravelStack(PopStack(S))
```

The proof rule for the loop is

$$\frac{P \supset A,\ A \wedge \sim B \supset Q,\ A \wedge B\{S\}A}{P\{\text{loop as } A \text{ while } B \text{ do } S \text{ repeat}\}Q}$$

The proof of this program is relatively easy using the standard inductive assertion method. Note that the only reasoning necessary or available for the proof is logic, the associativity property of ⊗, and "stack algebra" [Guttag76].

Version 2: Instantiation of the schema to tree traversal

Now, assume type D1 is a binary tree of the usual form, either nil or containing left and right subtrees, denoted Left(t) and Right(t), which are also trees. Let the type D2 be sequences, denoted ⟨...⟩ with catenation denoted ⊛. In the above schema, F can be instantiated to give a list of subtrees, either with or without nils, by respectively

Nodes(t)= if t=nil then ⟨⟩ else ⟨t⟩ ⊛ Nodes(Left(t)) ⊛ Nodes(Right(t))

or

Subtrees(t)= if t=nil then ⟨nil⟩ else ⟨t⟩ ⊛ Subtrees(Left(t)) ⊛ Subtrees(Right(t))

Here ⊛ is ⍟, the catenation operator between sequences
h0(t) is <>, the empty sequence, for Nodes and <nil> for Subtrees
h1(t) is <t>, h2(t) is Left(t), h3(t) is Right(t) for both Nodes and Subtrees
Therefore the following program computes Nodes

```
NodeList:=<>; t:=T; S:=CreateStack;
loop as Nodes(T)= NodeList ⍟ Nodes(t) ⍟ UnravelStack(S)
while t≠nil or S≠CreateStack do
(**) comment statement to be added here;
  if t≠nil then
    NodeList:= NodeList ⍟ <t>;
    S:=PushStack(S,Right(t)); t:=Left(t)
  else
(*)  NodeList:=NodeList ⍟ <>;
    t:=TopStack(S); S:=PopStack(S)
  fi
repeat
(*) NodeList:=NodeList ⍟ <>;
  assert Nodes(T)=NodeList;
```

The program for Subtrees differs only in the lines (*) as NodeList:=NodeList ⍟ <nil>.

Version 3: Inclusion of counting operations during traversal

Now suppose we want to traverse the tree in order to count something about its subtrees, say

leaf(t)= (t≠nil)∧Left(t)=nil∧Right(t)=nil

or

tip(t)= (t=nil)

We can do so by inserting C:=0 before the loop and make line (**), respectively

if t≠nil∧Left(t)=nil∧Right(t)=nil then C:=C+1 fi in the Nodes program
if t=nil then C:=C+1 fi in the Subtrees program

and proving the additional loop assertions

C=Count(NodeList)

where

Count(NodeList)=if NodeList=<> then 0
  else (if q(Last(NodeList)) then 1 else 0) + Count(OtherThanLast(NodeList))

with q respectively leaf and tip. For q as leaf, this gives

```
NodeList:=<>; t:=T; S:=CreateStack;
C:=0;
loop as Nodes(T)= NodeList ⍟ Nodes(t) ⍟ UnravelStack(S)
     ∧C=Count(NodeList)
while t≠nil or S≠CreateStack do
  if t≠nil∧Left(t)=nil∧Right(t)=nil then C:=C+1 fi;
  if t≠nil then
```

```
        NodeList:= NodeList ⍝ <t>;
        S:=PushStack(S,Right(t));   t:=Left(t)
      else
        NodeList:=NodeList ⍝ <>;
        t:=TopStack(S);   S:=PopStack(S)
    repeat
    NodeList:= NodeList ⍝ <>;
    assert Nodes(T)=NodeList  ∧  C=Count(NodeList)
```

Version 4: Optimization of the leaf-counting version

Now we will go through a long string of optimizing transformations. First, we move the counting operation inside the if-then-else

---

Part of program to be replaced

```
 P{ if B1∧B2 then S1 fi;
   if B1 then S2 else S3 fi }Q
==>
```

Replacing part

```
 P'{ if B1 then
      if B2 then S1 fi;
      S2;
    else S3 fi
 Q}
```

Sufficient conditions for correctness preservation

when P'∧B1∧B2 {S1} B1 that is, S1 does not change B1

---

because the t≠nil is preserved over C:=C+1

Next, observe that, stacks usually being finitely implemented and therefore subject to overflow, we might want to avoid putting nil Right(t) on the stack, so we introduce the statement if Right(t)≠nil then S:=PushStack(S,Right(t)) fi; using the transformation

---

P{S1}Q ==> P{if B then S1 fi}Q
when P'∧~B ⊃ Q

In this transformation, the strongest precondition for S1, P, is

```
      Nodes(T)=NodeList' ⍝ Nodes(t) ⍝ UnravelStack(S)
     ∧C'=Count(NodeList') ∧(t≠nil or S≠CreateStack)∧(t≠nil)
     ∧(C= if Left(t)=nil ∧ Right(t)=nil then C'+1 else C')
     ∧NodeList=NodeList' ⍝ <t>
```

B is Right(t)=nil and Q, the weakest necessary postcondition, is

Nodes(T)=NodeList ⊛ Nodes(Left(t)) ⊛ UnravelStack(S)
∧C=Count(NodeList)

The enabling condition holds because Nodes(t)=⟨t⟩ ⊛ Nodes(Left(t)) ⊛ Nodes(Right(t)) and Nodes(Right(t))=⟨⟩. This allows us to prove the additional invariant

NoNilNodesOnStack(S)= (S=CreateStack or
TopStack(S)≠nil∧NoNilNodesOnStack(PopStack(S)))

giving the result of all these transformations as

```
NodeList:=⟨⟩; t:=T; S:=CreateStack;
C:=0;
loop as Nodes(T)= NodeList ⊛ Nodes(t) ⊛ UnravelStack(S)
      ∧C=Count(NodeList)
      ∧NoNilNodesOnStack(S)
while t≠nil or S≠CreateStack do
  if t≠nil then
    if Left(t)=nil∧Right(t)=nil then C:=C+1 fi;
    NodeList:= NodeList ⊛ ⟨t⟩;
    if Right(t)≠nil then S:=PushStack(S,Right(t)) fi;
    t:=Left(t)
  else
    NodeList:=NodeList ⊛ ⟨⟩;
    t:=TopStack(S);      S:=PopStack(S)
  fi
repeat
 NodeList:=NodeList ⊛ ⟨⟩;
assert Nodes(T)=NodeList  ∧C=Count(NodeList)
```

Of course, NodeList is unnecessary in this program so, it can be deleted using the Ghost Variable Theorem [Gerhart78]

```
t:=T; S:=CreateStack;
C:=0;
loop as ∃ NodeList: ( Nodes(T)= NodeList ⊛ Nodes(t) ⊛ UnravelStack(S)
      ∧C=Count(NodeList) ∧NoNilNodesOnStack(S) )
while t≠nil or S≠CreateStack do
  if t≠nil then
    if Left(t)=nil∧Right(t)=nil then C:=C+1 fi;
    if Right(t)≠nil then S:=PushStack(S,Right(t)) fi;
    t:=Left(t)
  else
    t:=TopStack(S);      S:=PopStack(S)
repeat
assert ∃NodeList (Nodes(T)=NodeList  ∧C=Count(NodeList))
```

After proving the distributivity of Count over ⊛, i.e., Count(a⊛b)=Count(a)+Count(b), the assertions may be reworked to

Count(Nodes(T))=C + Count(Nodes(t)) + Count(UnravelStack(S))
and
C=Count(Nodes(T))
and the ghost variables are all gone. Finally, there are many redundant tests within the program, such as finding Left(t)=nil at the test before C:=C+1, making t be Left(t), then finding t nil in the next loop traversal, and PopStacking Right(old t) to become the new t, which can be shortened to t:=Right(t). Removing these redundant tests by twisting around the paths of the program, proving that the verification conditions for the new paths follow from those for the paths of the previous programs, leaves the more efficient, but uglier,

```
t:=T; S:=CreateStack; C:=0;
if t/nil then
   L: assert Count(Nodes(T))=C + Count(Nodes(t)) + Count(UnravelStack(S))
        ^ t/nil ^NoNilNodesOnStack(S);
   if Left(t)=nil then
     if Right(t)=nil then
       C:=C+1;
       if S=CreateStack then  goto Finish
       else  t:=TopStack(S); S:=PopStack(S);  goto L fi
     else  t:=Right(t); goto L  fi
   else
     if Right(t)/nil then S:=PushStack(S,t); t:=Left(t); goto L
     else  t:=Left(t); goto L  fi
   fi;
fi;
Finish: assert Count(Nodes(T))=C;
```

Notice some of the characteristics of this program derivation:

(1) It is formally controlled. If a program gets into the wrong form, if an idea occurs for a new and better form, or if there is simply a need to step up to more complex programs, then there is a bridge to systematically transfer correctness from the old to the new form, which seldom requires much new proof. However, the process is tedious and requires considerable program rewriting.

(2) The assertions have a clean structure which breaks into various parts:
Nodes(T)=... the dominant clause, describing the goal of the program
NoNilNodesOnStack(S) ... a property which assisted a space optimization
C=Count(NodeList)... relates a concrete value C to an abstract value NodeList

t/nil... a special condition picked up at the loop start

From these, we can conjecture some possible "laws" of assertions:

(a) Ghost variables represent missing abstractions,

(b) Assertion clauses may be classified as dominant, by association with the least optimized abstract program, or optimizational, by association with some property used for optimization,

(c) Assertion clauses may be proved one by one in some strategic order,

(d) Many assertions have the form FinalResult=Current ...YetToBeDone because they originate from functions.

(3) The versions 1 and 2 can be reused for other problems and other orders of tree traversal may be modeled after this one. The price for this generality is that the instantiated schemas must be optimized. There is a tradeoff between finding the optimizing transformations and the supporting assertion increments and finding the fully optimized final program and the assertions for it.

(4) There are numerous questions about this approach:

(a) Are the laws valid? useful? (How do we decide this?)

(b) How can the tedium of managing multiple versions be reduced?

(c) How hard is it to find schemas? Are they worth the effort? What level of abstraction provides the greatest payoff? For example, is there a generalization of Version 1 from which tree traversal is a direct instantiation?

(d) How hard is it to maintain a catalog of schemas and transformations?

REFERENCES

[Ambler77] Ambler, A. L., D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch, and R. E. Wells, "Gypsy: A language for specification and implementation of verifiable programs," *Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices*, 12 (3), March 1977, 1-10.

[Balzer76] Balzer, R., N. Goldman, and D. Wile, "On the transformational implementation approach to programming," *2nd International Conference on Software Engineering*, October 1976.

[Bledsoe74] Bledsoe, W. W. and P. Bruell, "A man-machine theorem-proving system," *Artificial Intelligence*, 5 (1), Spring 1974, 51-72.

[Boyer75] Boyer, R. S., and J. S. Moore, "Proving theorems about Lisp functions," *J. ACM*, 22 (1), January 1975, 129-144.

[Boyer77] Boyer, R. S., and J S. Moore, "A lemma driven automatic theorem prover for recursive function theory," *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, Boston, Massachusetts, August 1977, 511-519.

[Burstall69] Burstall, R.W., "Proving properties of programs by structural induction," *The Computer Journal*, 12 (1), 1969, 41-48.

[Burstall74] Burstall, R.W., "Program proving as hand simulation with a little induction," *Information Processing 74*, Stockholm, Sweden, 1974, 308-312.

[Burstall77a] Burstall, R.W. and J. Darlington, "A transformation system for developing recursive programs," *J. ACM*, 24(1), January 1977, 44-67.

[Burstall77b] Burstall, R., and J. Goguen, "Putting theories together to make specifications," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Boston, Massachusetts, August 1977, 1045-1058.

[Carter77] Carter, W. C., H. A. Ellozy, W. H. Joyner, Jr., G. B. Leeman, Jr., "Techniques for microprogram validation," IBM, T.J. Watson Research Center Report RC 6361.

[deBakker77] deBakker, J.W., "Semantics and the foundations of program proving," *Information Processing 77*, Toronto, Canada, August 1977, 279-284.

[Dijkstra74] Dijkstra, E. W., "Programming as a discipline of a mathematical nature," *American Mathematical Monthly*, June 1974, and *Computers and People*, October 1974.

[Dijkstra76] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, 1976.

[Elspas77] Elspas, B., R. E. Shostak, and J. M. Spitzen, *A verification system for Jocit/J3 programs (Rugged programming environment - RPE/2)*, Stanford Research Institute and Rome Air Development Center Technical Report RADC-TR-77-229, June 1977.

[Gerhart76] Gerhart, S., "Proof theory of partial correctness verification systems," *SIAM Journal of Computing*, 5 (3), September 1976, 355-377.

[Gerhart78] Gerhart, S., "Two proof techniques for transferral of program correctness," (forthcoming).

[Good77] Good, D. I., *Constructing verifiably reliable and secure communications processing systems*, Institute for Computing Science and Computer Applications Report ISCSA-CMP-6, The University of Texas at Austin, January 1977.

[Goodenough77] Goodenough, J. and S. L. Gerhart, "Toward a theory of testing: data selection criteria," in *Current Trends in Programming Methodology*, Vol. II, *Program Validation*, R. T. Yeh (ed.), Prentice-Hall, 1977, 44-79.

[Guttag76] Guttag, J. V., E. Horowitz, and D. R. Musser, *Abstract data types and software validation*, Information Sciences Institute, ISI/RR-76-48, August 1976.

[Hoare73] Hoare, C. A. R., and N. Wirth, "An axiomatic definition of the programming language Pascal," *Acta Informatica*, 2 (4), 1973, 335-355.

[Howden76] Howden, W. E., "Reliability of the path analysis testing strategy," *IEEE Transactions on Software Engineering*, September 1976.

[Lampson77] Lampson, B. W., J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek, "Report on the programming language Euclid," *SIGPLAN Notices*, 12 (2), February 1977.

[Lankford77] Lankford, D. S. and A. M. Ballantyne, *Decision Procedures for Simple Equational Theories with Commutative-associative Axioms: Complete Sets of Commutative-associative Reductions*, University of Texas Automatic Theorem Proving Project Report ATP-39, August 1977.

[Levin77] Levin, R., *Program Structures for Exceptional Condition Handling*, Ph.D. thesis, Carnegie-Mellon University, 1977.

[Liskov77] Liskov, B. H., A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in Clu," *Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices*, 12 (3), March 1977, 166-178. Also *Comm. ACM*, 20 (8), August 1977, 564-576.

[London77a] London, R. L., "Perspectives on program verification," in *Current Trends in Programming Methodology*, Vol. II, *Program Validation*, R. T. Yeh (ed.), Prentice-Hall, 1977, 151-172.

[London77b] London, R. L., J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek, "Proof rules for the programming language Euclid," *Acta Informatica*, 1978 (to appear).

[London77c] London, R. L., "Program verification," in *Research Directions in Software Technology*, Peter Wegner (ed.), (to appear).

[Luckham76] Luckham, D. C. and N. Suzuki, "Automatic program verification V: Verification-oriented proof rules for arrays, records and pointers," Stanford University Artificial Intelligence Laboratory Memo AIM-278, March 1976.

[Luckham77] Luckham, D. C., "Program verification and verification oriented programming," *Information Processing 77, Proceedings of IFIP Congress 77*, B. Gilchrist (ed.), North-Holland, 1977, 783-793.

[Manna76] Manna, Z. and R. J. Waldinger, "Is 'sometime' sometimes better than 'always'? Intermittent assertions in proving program correctness," *Proceedings of the Second International Conference on Software Engineering*, October 1976, 32-39. Also *Comm. ACM*, 21, 2, February 1978.

[Manna77] Manna, Z. and R. Waldinger, "Synthesis: dreams => programs," SRI International Technical Note 156, November 1977.

[Moriconi77] Moriconi, M., *A system for incrementally designing and verifying programs*, Ph.D. thesis, University of Texas, Austin, November 1977. Also

ISI/RR-77-65 (Vol. 1) and ISI/RR-77-66 (Vol. 2), Information Sciences Institute, November 1977.

[Morris77] Morris, J. H., Jr. and B. Wegbreit, "Subgoal induction," *Comm. ACM*, 20 (4), April 1977, 209-222.

[Musser77] Musser, D. R., "A data type verification system based on rewrite rules," *Sixth Texas Conference on Computing Systems*, Austin, Texas, November 1977.

[Nelson78] Nelson, G. and D. C. Oppen, "A simplifier based on efficient decision algorithms," *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978.

[Owicki76] Owicki, S. S. and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Comm. ACM* 19 (5), May 1976, 279-285.

[Pratt78] Pratt, V. and D. Harel, "Nondeterminism in logics of programs," *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978.

[Suzuki77] Suzuki, N. and K. Ishihata, "Implementation of an array bound checker," *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, January 1977, 132-143.

[Wegbreit74] Wegbreit, B., "The synthesis of loop predicates," *Comm. ACM*, 17 (5), 1974, 251-264.

[Wegbreit77] Wegbreit, B., "Constructive methods in program verification," *IEEE Transactions on Software Engineering*, SE-3 (3), May 1977, 193-209.

[Wulf76] Wulf, W. A., R. L. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs," *IEEE Transactions on Software Engineering*, SE-2 (4), December 1976, 253-265. See also Carnegie-Mellon University and Information Sciences Institute Technical Reports, 1976.